Group Members:

Tanveer Ahmed (FA20-BCS-063)

Usama Pervez(SP20-BCS-013)

Project Report: Simple Lexical Analyzer for Basic Programming Language

Introduction:

The goal of this project was to implement a simple lexical analyzer for a basic programming language using C#. The lexical analyzer is a crucial component in the compiler construction process, responsible for tokenizing the source code into meaningful units for further processing.

Project Objectives:

Tokenization:

Implement a lexer to recognize and tokenize identifiers, numeric literals, and basic arithmetic operators.

Keyword Recognition:

Identify and distinguish keywords, such as the assignment keyword ("let").

Error Handling:

Implement robust error handling to gracefully handle invalid characters and scenarios.

Modularity:

Design the code with modularity in mind, separating concerns between different components for improved maintainability.

User Interaction:

Provide a simple user interface for users to input code and observe the lexer's tokenization process.

Challenges Faced

Output Display Issue:

Initially faced trouble displaying the output of the lexical analyzer.

Resolved by addressing a variable naming discrepancy (Semicolon vs. EndLine) and handling null values in the Console.WriteLine statement.

Missing SemicolonEnum:

Discovered the absence of the Semicolon token type in the TokenType enum.

Addressed by adding Semicolon to the enum for proper handling of semicolons in the input code.

Exception Handling:

Identified a need for improved exception handling to provide more informative error messages.

Specifically, addressed the handling of invalid characters by including details about the problematic character and its position in the input code.

Modularization:

Recognized the importance of further modularization for better code organization.

Worked on breaking down the lexer functionality into smaller, more focused methods to enhance readability and maintainability.

Token Naming Consistency:

Ensured consistency in token naming by adding the missing Semicolon token type to maintain uniformity in the code.

User Interface and Interaction:

Considered potential future enhancements related to user interaction, such as accepting input from a file and improving the overall user experience.

Testing Approach:

Emphasized the need for a more robust testing approach, including thorough testing with various inputs, edge cases, and invalid inputs.

Output Format Improvement:

Discussed the possibility of modifying the output format to make it more informative, including line numbers and positions for tokens in the input code.

Conclusion:

The project successfully implemented a simple lexical analyzer, addressing challenges related to output display, missing token types, exception handling, modularity, and potential improvements in user interaction and testing. The resulting lexer provides a foundational component for further stages in the compiler construction process.

Recommendations for Future Work

Enhanced User Interface:

Consider extending the user interface to provide additional features, such as file input and a more interactive experience.

Code Optimization:

Explore opportunities for code optimization to improve the efficiency of the lexer.

Integration with Parser:

Integrate the lexical analyzer with a parser for more advanced language processing.

Documentation:

```
}
```

Expand documentation to provide comprehensive information about the lexer's functionality and usage.

CODE:

```
using System;

namespace hiovfshgoih

{
    public class Token
    {
        public TokenType Type { get; }
        public string Value { get; }

        public Token(TokenType type, string value)
        {
            Type = type;
            Value = value;
        }
    }
```

```
public enum TokenType
{
    Identifier,
    IntegerLiteral,
    FloatLiteral,
    Assignment,
    Plus,
    Minus,
    Multiply,
    Divide,
    LeftParen,
    RightParen,
    Semicolon,
    EndOfFile
}

public class Lexer
{
    private readonly string input;
    private int currentPosition;
```

```
        public Lexer(string input)

        {

            this.input = input;

            currentPosition = 0;

        }



        private char CurrentChar => currentPosition < input.Length ?
input[currentPosition] : '\0';



        private void MoveNext()

        {

            currentPosition++;

        }



        private void SkipWhitespace()

        {

            while (char.IsWhiteSpace(CurrentChar))

            {

                MoveNext();

            }

        }
```

```
private bool IsDigit(char c)
{
    return char.IsDigit(c) || c == '.';
}

private Token ReadIdentifierOrKeyword()
{
    string identifier = "";
    while (char.IsLetterOrDigit(CurrentChar) || CurrentChar == '_')
    {
        identifier += CurrentChar;
        MoveNext();
    }

    // Check if the identifier is a keyword
    if (identifier.Equals("let", StringComparison.OrdinalIgnoreCase))
    {
        return new Token(TokenType.Assignment, "=");
    }
    return new Token(TokenType.Identifier, identifier);
}

private Token ReadNumber()
{
    string number = "";
    while (IsDigit(CurrentChar))
    {
        number += CurrentChar;
        MoveNext();
    }

    if (number.Contains("."))
    {
        return new Token(TokenType.FloatLiteral, number);
    }
    else
    {
        return new Token(TokenType.IntegerLiteral, number);
    }
}
```

```
public Token GetNextToken()
{
    SkipWhitespace();

    if (currentPosition >= input.Length)
    {
        return new Token(TokenType.EndOfFile, "");
    }

    char currentChar = CurrentChar;

    switch (currentChar)
    {
        case '+':
            MoveNext();
            return new Token(TokenType.Plus, "+");

        case '-':
            MoveNext();
            return new Token(TokenType.Minus, "-");
```

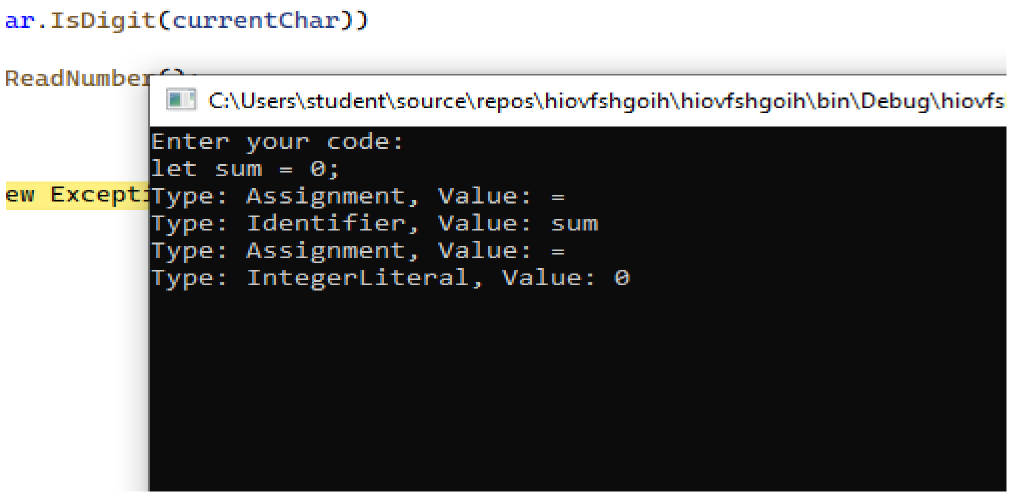
```
case '*':
    MoveNext();
    return new Token(TokenType.Multiply, "*");

case '/':
    MoveNext();
    return new Token(TokenType.Divide, "/");

case '(':
    MoveNext();
    return new Token(TokenType.LeftParen, "(");

case ')':
    MoveNext();
    return new Token(TokenType.RightParen, ")");

case '=':
    MoveNext();
    return new Token(TokenType.Assignment, "=");
```

```
            case ';':
                MoveNext();
                return new Token(TokenType.Semicolon, ";");

            default:
                if (char.IsLetter(currentChar) || currentChar == '_')
                {
                    return ReadIdentifierOrKeyword();
                }
                else if (char.IsDigit(currentChar))
                {
                    return ReadNumber();
                }
                else
                {
                    throw new Exception($"Invalid character: {currentChar}");
                }
        }
    }
}
```

```
class Program

{

    static void Main()

    {

        string inputCode="gdhgfdhb";

        for (int i = 0; i > -1; i++)

        {

            Console.WriteLine("Enter your code:");

             inputCode = Console.ReadLine();

            Lexer lexer = new Lexer(inputCode);


            Token token;

            do

            {

                token = lexer.GetNextToken();

                Console.WriteLine($"Type: {token.Type}, Value: {token.Value}");

            } while (token.Type != TokenType.EndOfFile);

        }
```

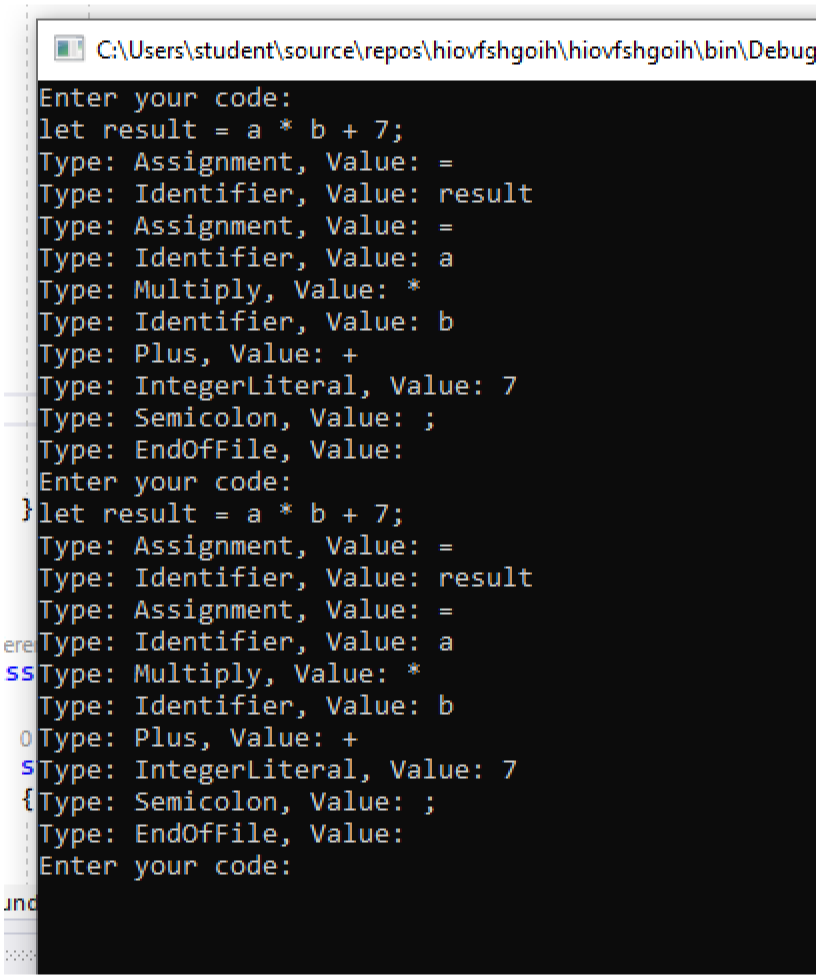
```
            Console.WriteLine("Lexical analysis complete.");
        }
    }
}
```

OUTPUT:



```
ar.IsDigit(currentChar))
ReadNumber();
ew Excepti
```

```
C:\Users\student\source\repos\hiovfshgoih\hiovfshgoih\bin\Debug\hiovfs
Enter your code:
let sum = 0;
Type: Assignment, Value: =
Type: Identifier, Value: sum
Type: Assignment, Value: =
Type: IntegerLiteral, Value: 0
```


C:\Users\student\source\repos\hiovfshgoih\hiovfshgoih\bin\Debug

```
Enter your code:
let result = a * b + 7;
Type: Assignment, Value: =
Type: Identifier, Value: result
Type: Assignment, Value: =
Type: Identifier, Value: a
Type: Multiply, Value: *
Type: Identifier, Value: b
Type: Plus, Value: +
Type: IntegerLiteral, Value: 7
Type: Semicolon, Value: ;
Type: EndOfFile, Value:
Enter your code:
let result = a * b + 7;
Type: Assignment, Value: =
Type: Identifier, Value: result
Type: Assignment, Value: =
Type: Identifier, Value: a
Type: Multiply, Value: *
Type: Identifier, Value: b
Type: Plus, Value: +
Type: IntegerLiteral, Value: 7
Type: Semicolon, Value: ;
Type: EndOfFile, Value:
Enter your code:
```